Chapter 7

Design of an OO/FV Library

In the last few years an object oriented numerical library has been developed. The main objective was to gather work, which would be done in different projects, into a single piece of software. This should enable people to share the benefits of new developments. Many numerical techniques can be applied to different problems. A computational grid, for example, is needed by almost all sorts of numerical applications. Work in the last years showed that it was possible to incorporate experiences and developments from various numerical projects into this software. The software design had to fulfill several design criteria:

- a reliable programming interface
- reasonable efficiency
- maximal reuse-ability of components
- extendible
- customizable to various problems

To find a stable programming interface proved to be a non-trivial task. In the beginning of the development the original design had to be altered and refined several times. After this first initial phase, however, the software structure became stable. Only minor modifications were needed from then on. By talking about a 'reasonable' efficiency it is meant, that a tradeoff had to be found between a clear and object-oriented structure and plain performance aspects. As it has already been outlined in chapter 6 it was possible to find a good compromise, by using the template mechanism of the *C++* programming language. The main objective of the finite volume library, developed over the last few years, was to be a flexible toolbox. Therefore computational efficency was not the single most important design objective, although it is an important point, of course. Another main criterion was, that new physical problems can be tackled quickly. For this reason it seems reasonable to make minor concessions in terms of CPU efficiency. One option is of course to highly optimize critical parts of the program, without altering the programming interface for new developments. The final design, which has been found, offers both, an

```
PrintText {This~is~a~simple~loop~example.}
StepAction {
  PrintStepCounter {}
  5
}
PrintText {Now~the~run~is~finished.}

MOUSE is using port 1111

no errors found in the control file

This is a simple loop example.
step counter = 1
step counter = 2
step counter = 3
step counter = 4
step counter = 5
Now the run is finished.
```

Listing 7.1: A very basic *MCL* program.

optimizable code and a flexible interface. Furthermore it gives programmers the opportunity to hand-optimize critical parts, without effecting the overall software structure. Thus new developments can be prototyped quickly and computational efficiency would already be acceptable. If, however, a module becomes very frequently used, the effort to hand-optimize might pay off. Basically the library consists of three different levels now:

1. the actual application (run by an *end-user*)
2. an interpreted control language
3. the underlying *C++* class library

There is a number of GUI tools and classes available which enable users to interactively run and test numerical applications. The intermediate layer of a control language offers the flexibility to customize applications. Thus it would be possible to have a graphical version and a parallelized batch version of the same numerical application, without recompiling any *C++* code. Finally the control language can be extended, by introducing new *C++* classes to the library, or by adding additional libraries to the project.

7.1 General Concepts

This section intends to briefly introduce the general programming concepts and interfaces of the developed finite-volume toolbox. *MOUSE* has borrowed some ideas from object oriented user interface programming. A running application consists of several objects, which interact with each other in different ways. As it is done in GUI programming, the objects are arranged in a hierarchical structure. Furthermore they can communicate,

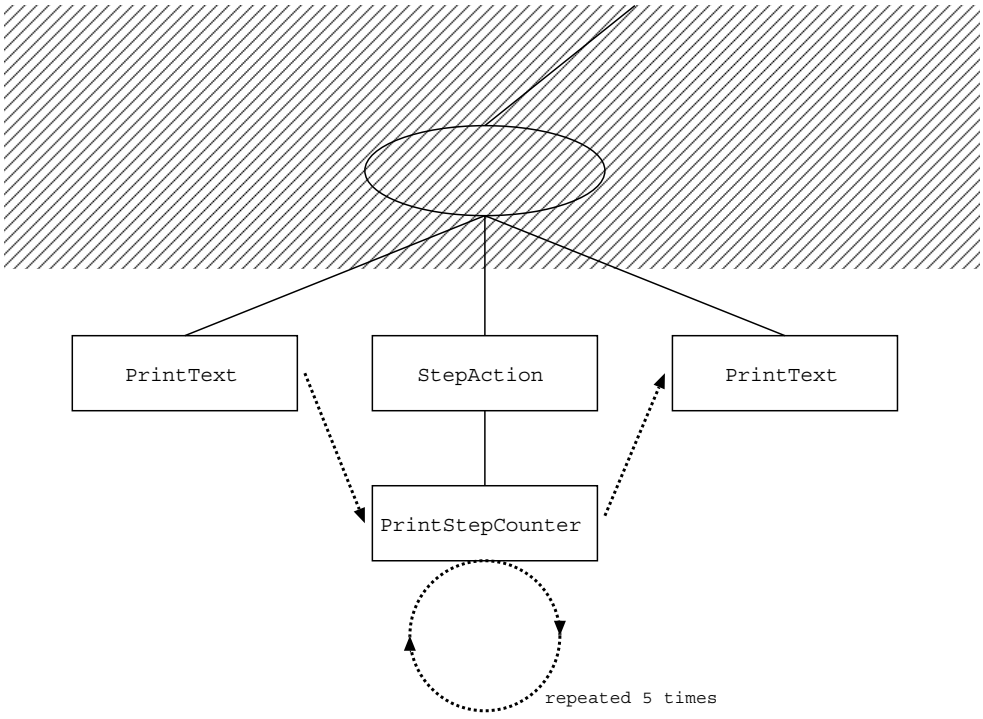

Figure 7.1: Object tree created by listing 7.1.

using an event mechanism very similar to what is known from GUI libraries. The main interface between the *end-user-application* and the more technical *C++* implementation level, is a control language. This language is called *MCL* and is an interpreted, script like, language. Its key idea is, that every *MCL* -command refers to a *C++* class. When using a command, the instantiation of an object of a particular class will be triggered. A typical run of a *MOUSE* application consists of two distinct passes:

1. parsing of a control file
2. execution of the specified commands

Listing 7.1 shows a simple example and the output it creates when run. The message *no errors found in the control file* indicates, that the first pass has been completed successfully. This two pass mechanism shall be explained in detail later. Figure 7.1 shows the object tree which will be generated in this case. Dotted arrows show the executing sequence of this small program. The hatched region indicates, that there are more objects on top of this small sample tree, which do not show up in the *MCL* code. These objects are of an internal nature and are mainly used for startup and communication purposes.

7.1.1 Main Programming Interfaces

As every object oriented library this library defines a number of abstract interfaces. The most important interfaces for the finite-volume library, described here, will be briefly

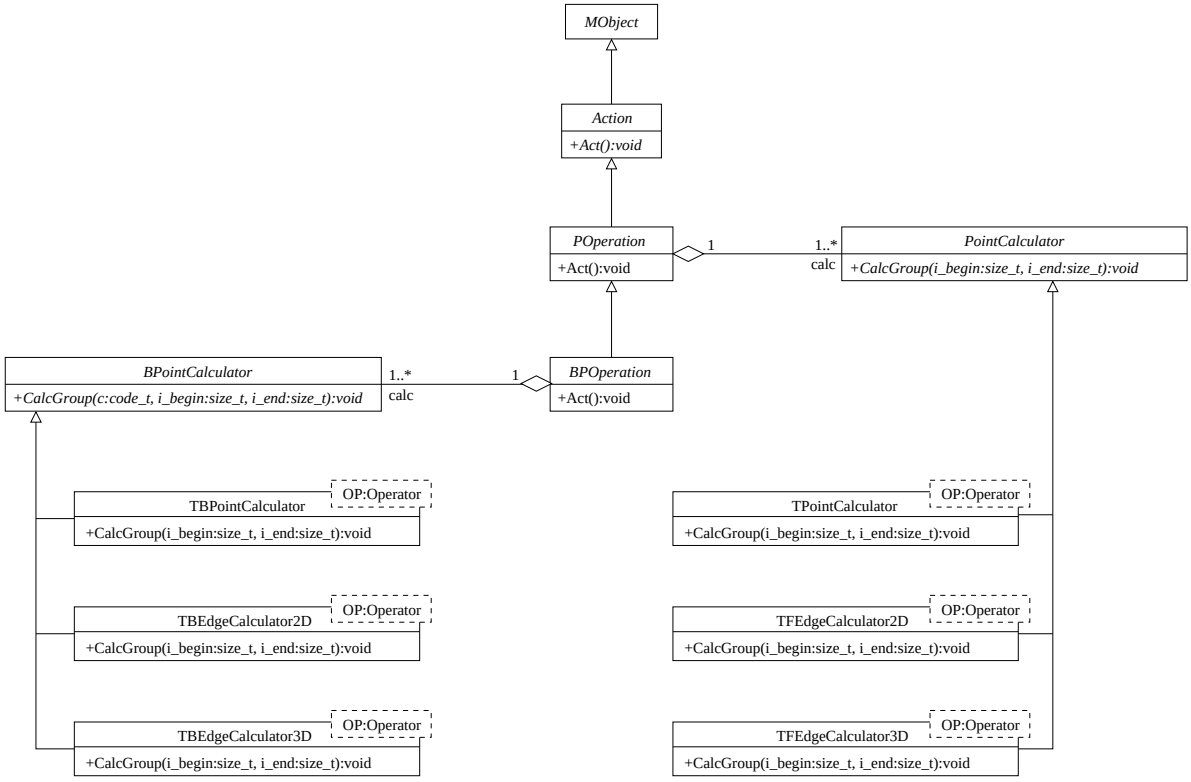

Figure 7.2: Class diagram for standard operators and calculators.

explained now. Of course this can only be a very rough overview. All visualization interfaces have been left out completely. Only the most essential ones are explained here, in order to give an idea how the library works and how it could eventually be extended. Possible extensions and new developments have always been in the mind of the developers.

MObject

`MObject` is the base-class for almost everything within the library. The only exceptions are some very basic data-storage classes. All functionality which is needed to create and handle the object-tree can be found in here. Furthermore it contains a few tools to write error-messages and simplify debugging. It also offers the necessary tools to use the event communication mechanism. A class `Event` exists, which can be created and send by programmers. This mechanism can for instance be used to find certain data-structures in the object-tree. Before the object-tree, based on `MObject`, has been introduced into the software, another approach had been used. Assume the following small example: A class shall be created to perform a simple one-step time-integration. The first thing to be done is to evaluate the flux-integral. After that the new time-level can be computed. Furthermore one might want to apply pointwise boundary conditions after the advancing step. Having this in mind, the class will probably be designed with two abstract interfaces, one for the flux integral and the other one for the boundary conditions. Using computational fluid dynamics as example this will work fine and different flux formulations (e.g. AUSM, Roe,

Roe-Turkel ...) can be combined with different boundary conditions. Nothing has to be rewritten for the time-integration method. After a while, however, one might want to extend it to viscous computations. Thus another interface has to be introduced to allow the usage of different formulations for the viscous fluxes. Even later it might happen, that chemical reactions need to be integrated into the solver. Once again more interfaces have to be introduced. This formulation is not satisfying, because it needs redesign on a regular basis. It might work for commercial codes, but in a research environment it is not acceptable, since altering the methods and trying different approaches is daily business there. Thus the open and flexible object-tree design has been chosen after some bad experiences with the more static approach in the very beginning of the project. Now it is a lot easier. The time-stepping class needs two interfaces, one is what has to be done before advancing in time and the other what has to be done afterwards. What is "plugged" into those interfaces is defined using the control-language. This can be almost everything, since the control-language is easily extendible. `MObject`s use a special constructing mechanism, which will be explained in paragraph 7.1.3.

Executable *MCL* Commands

`Action` is the base for every class that is intended to represent an executable control language command. The interface is most simple. An abstract method **void** `Act()` exists, that has to be overwritten by derived classes.

Calculators and Operators

MOUSE uses a method to combine fine-grained operations into groups of them, in order to be accessible via runtime-binding (please see also section 6.5.3). The key idea is that people can concentrate on what has to be done on the level of computational molecules. For example it is enough to write the flux from one node to another, if a new flux-splitting scheme shall be implemented. To keep a good CPU-efficiency those molecular operations will be addressed, using static polymorphism. A class named `PointCalculator` is the actual interface between these operations and the control-language. A few examples on how new operations can be implemented, will be introduced later. There are a number of different operator types available:

- `PointOperator`
  is not an actually existing class. It is rather a description how an operator should look like. For performance reasons no dynamic binding is used for such fine-grained operations. The *C++* template mechanism is used to realize static polymorphism instead. The same applies to the four following interface descriptions.

- `FEdgeOperator2D`
  is a generic operation along a two-dimensional field edge. Usually it will modify data on both adjacent nodes of the edge. The most common example probably is the evaluation of fluxes.

- `FEdgeOperator3D`
  is the three-dimensional counterpart of the above 2D-operator.

- `BEdgeOperator2D`
  describes a two-dimensional operation across boundary edges.

- `BEdgeOperator3D`
  is a three-dimensional operation across boundaries. This operator, however, varies significantly from the 2D version.

7.1.2 Data Storage Classes

Furthermore the library offers a variety of generic and specialized data-structures. Figure 7.3 shows a class-diagram of the list data-structures in use. The diagram is incomplete, since it is only intended to give a brief overview. Most of the classes are shown, but the operations and attributes are mostly skipped, only a few key-operations are displayed. Following are the main data-structures of the library:

A Base for One-Dimensional Structures (`List`)

`List` is the abstract base-class for all container-classes. It provides the ability to grow and shrink. Entries can be created and deleted. To optimize performance, `List` is able to handle "holes". They appear where existing entries have been discarded. The `List`-kernel keeps track of the deleted entries and tries to fill the created holes, whenever new entries are allocated. Furthermore it offers a cleanup procedure. Therefore a copy routine for single entries has to be provided by derived classes. To handle loops over lists with such holes special iterating methods are provided. These are very simple and can be easily inlined by the compiler. Thus the resulting performance penalty is minimal, if at all measurable. Tests on PC's running *Linux* as operating system and using the *GNU C++* compiler, however, showed no performance loss at all. A `List` has an initial size when it is created. If later on more entries are allocated, than the original size permits, the list will be extended. This, however, is a fairly costly operation, since the whole list has to be moved to a new location in the computer's memory. For this operation the same copy routine will be used as for the cleanup procedure. Another important feature of this data-type is that it has the capability of being linked to other lists of the same base-type. Thus they behave synchronously (i.e. they have exactly the same length and all possible "holes" are in the same positions). Since all major data-structures are directly or indirectly derived from `List` this offers a powerful tool. `TList` is a simple generic version of `List`. There is no requirement for the data-type to be stored in it. Every valid data-type can be used as parameter for this template-class.

Sparse Two-Dimensional Structures (`SparseTwoDimArray` and Derived Classes)

`SparseTwoDimArray` and derived classes provide tools for storing sparse data. An example for data stored in such a structure are the neighborship relations of a mesh. A specialized

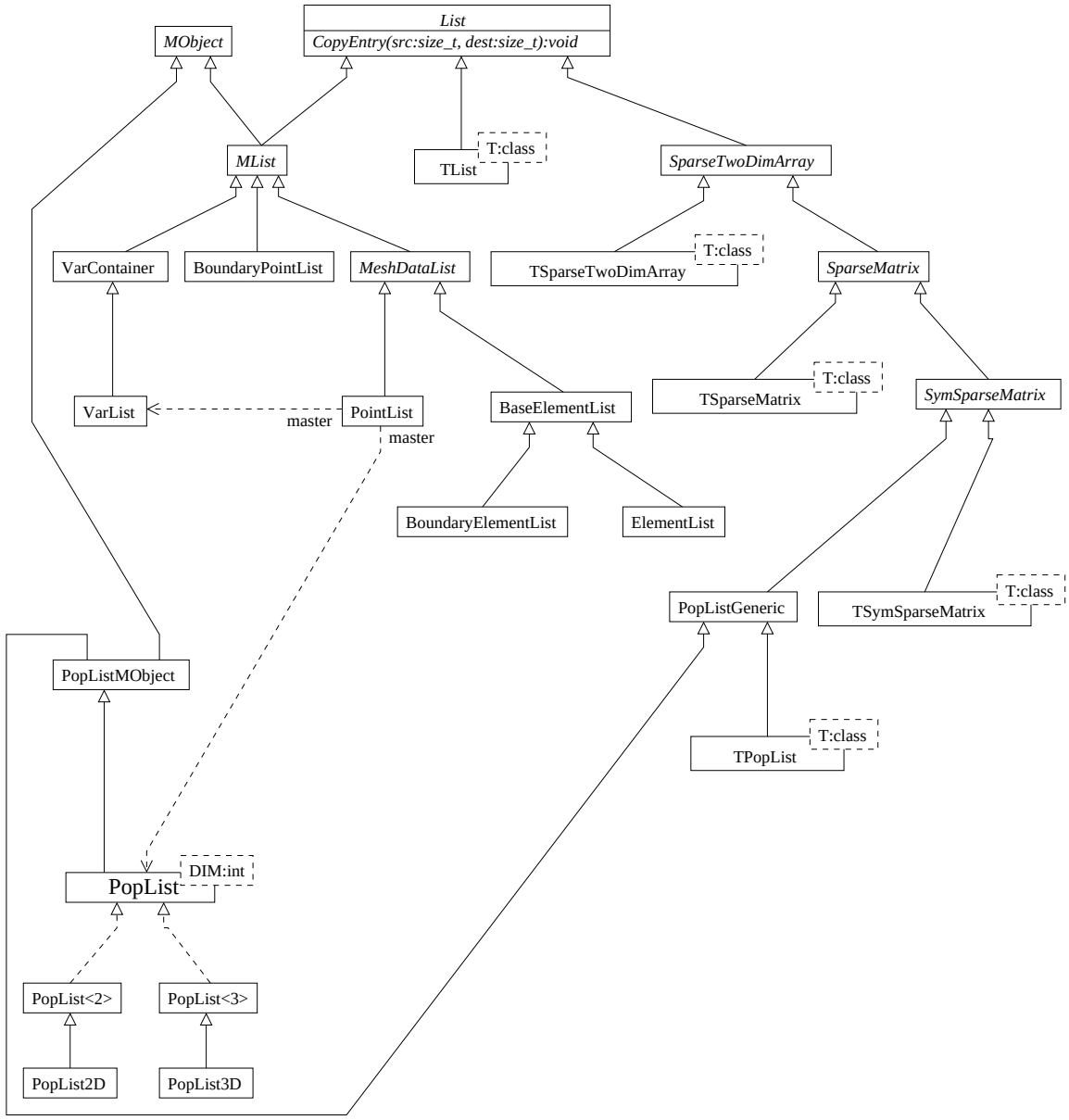

Figure 7.3: Class diagram for data storage classes.

data-structure `PopList` (*pop* is an abbreviation for **p**oint to **o**ther **p**oint) exists, which is derived from this sparse structure (see also the next paragraph, please). Another example of sparse data would be the matrix representing a system of linear equations (e.g. resulting from an implicit discretization). More specialized versions (`SparseMatrix` and `SymSparseMatrix`), as well as generic versions (`TSparseTwoDimArray`, `TSparseMatrix` and `TSymSparseMatrix`) exist. For the sparse data-structures a similar linking mechanism exists as it does for simple one-dimensional lists. The second dimension will of course extend and shrink simultaneously, if sparse data-structures are linked together. This linking mechanism proved to be an efficient development tool.

Point to Other Point Structures

`PopListGeneric` and its derived classes are sparse data structures that represent a graph of an unstructured grid. For every node of a mesh information about all neighboring nodes is provided. Special routines exists to create such structures from different sources. It is also possible to easily create new sources for this neighboring information, but this shall not be explained in the scope of this text. Currently it is possible to create the neighbor information from the following sources:

- 2D and 3D point and element information
- 2D and 3D structured mesh information

`PopList2D` and `PopList3D` are the main data structures to perform edge-oriented computations (e.g. flux integral evaluations).

Global Structures for Mesh Related Data

`MList` and its child class `MeshDataList` offer a combination of `MObject` and `List` and thus the connection to the control-language. A number of specialized classes for different purposes in a mesh based computation exist. All of them represent key words in *MCL* . There are, however, not executable because they represent data-storage objects. Just like an **int** in *C++* would not be executable. Thus you would never write

```
...
PointList{10000 1000 2 test yes}
...
```

to create a 2D `PointList` with an initial size of 10000 nodes and an increment of 1000 nodes. The keyword `yes` is a boolean argument and in this case means that the `PointList` shall be read from the file `"test.points"`. If written in this way the interpreter will try to execute it, which cannot work since `PointList` is nor directly neither indirectly derived from `Action`. Correctly it should look like this:

```
...
CreateObject{PointList{10000 1000 2 test yes}}
...
```

or

```
...
CreateObject{%PL PointList{10000 1000 2 test yes}}
...
```

if a pointer should be kept on the created object. `CreateObject` can be understood as a variable declaration in *MCL* . The following global structures are part of the library:

- `PointList`
- `BoundaryPointList`

- `BaseElementList`
- `ElementList`
- `BoundaryElementList`

VarContainer

`VarContainer` is a data-structure with three levels of indexing. It has been designed to store numerical field-data (e.g. the conservative variables). The data-type of this container class is `real`. The first two levels can be accessed symbolically and numerically, whereas the third level can only be accessed numerically. The different indexing levels are:

1. a field-name (e.g. "old" and "new" representing an old and a new time-level)
2. a variable name (e.g. "rho", "rhou", ... in a compressible-gas-flow computation)
3. a numerical index (e.g. a node-index if the `VarContainer` is linked to the `PointList` of a mesh)

A derived class `VarList` was created which is always linked to the computation's `PointList`. Thus the third index will always be the index of a node in the mesh. `VarContainer` has two nested types (`var1_t` and `var2_t`) which represent automatically updated pointers to the stored numerical data.

- `var1_t` is a one-dimensional pointer. Assuming that the variable `VL` is a pointer to a `VarList`, the following code-snippet would assign an automatic pointer:

  ```
  ...
  VarContainer::var1_t var = VL->Get1DVar("new","rho");
  ...
  ```

  Now `var` is a pointer to the density-data of the field named "new".

- `var2_t` is a two-dimensional pointer, which offers the opportunity to perform a loop over all variables in a field. Thus the second index becomes numerical. This is important, since many operations are independent of the set of variables used (e.g. copy operations or the computation of numerical gradients).

These automated pointers proved to be very useful, since they are always usable no matter how many entries have been created or deleted. In principle the pointers can be completely inlined by the compiler. Tests with the *GNU* compiler showed no performance penalty.

```
class MObject
{
  ...
public:
  MObject (carg_t arg);
  template <class T> T* BuildObject (T *aT, carg_t arg);
  ...
}

class A : public MObject
{
  int a1, a2;
public:
  A (carg_t arg);
}

class B : public MObject
{
  int b1;
public:
  B (carg_t arg);
}

class C : public B
{
  real c1;
  string c2;
  A *c3, *c4;
public:
  C (carg_t arg);
}
```

Listing 7.2: Simple example to illustrate constructor mechanism.

7.1.3 Constructor Call Mechanism

The control language is used to create an object-structure, which will later on define the execution sequence of the programs. Basically an *MCL* command will trigger a constructor-call. All objects which are created, using this scripting-language, have to be of a common base class (`MObject`). This base class contains the functionality needed to handle large object-tree-structures. A control-command has the following structure:
`$VAR_NAME CLASS_NAME {...}`

- `$VAR_NAME` This is optional and specifies a variable, which may be used later to refer to the created object. If the first character found is a "$" the word will be interpreted as variable, otherwise this will be skipped and it will be interpreted as class name.
- `CLASS_NAME`
  This specifies the class of the object to create. The class must have been derived

```
A::A(carg_t arg) : MObject(arg)
{
  AddClassName();
  a1 = arg.P->ReadInt()
  a2 = arg.P->ReadInt()
};

B::B(carg_t arg) : MObject(arg)
{
  AddClassName();
  b1 = arg.P->ReadInt();
};

C::C(carg_t arg) : B(arg)
{
  AddClassName();
  c1 = arg.P->ReadReal();
  c2 = arg.P->ReadString();
  c3 = BuildObject(c3, arg);
  c4 = BuildObject(c4, arg);
};
```

Listing 7.3: Constructor function bodies. (see listing 7.2)

from `MObject` in some way.

- `{ ... }`
  Everything enclosed in the brackets will be passed to the constructor of this class.

A small and abstract example shall be used now to describe the interpretation process of a command. Listing 7.2 shows three example classes. Class `C` represents an *MCL* command and the following line could be used to create an object of this type, using the control-language.

```
CreateObject{$A_C C{7 10.8 text A{1 2} A{5 7}}}
```

`carg_t` which is used as argument for the constructors is an alias for `ConstructArg`. This type is the default argument for any class derived from `MObject`. It contains information about the parent in the object-tree, as well as a pointer to a `Parser` which is used to read the control-script. Figure 7.4 shows the object and class relations which result from this abstract example. To examine the construction process a close look at the constructor of class `C` (see listing 7.3, line 14) is helpful. `arg.P` is a pointer to the `Parser` used and it contains this text:

```
7 10.8 text A{1 2} A{5 7}
```

First of all the argument `arg` will be passed to the constructor of the direct base class `B` (see listing 7.3, line 8). `arg.P` still contains the same text here. The first command in the constructor

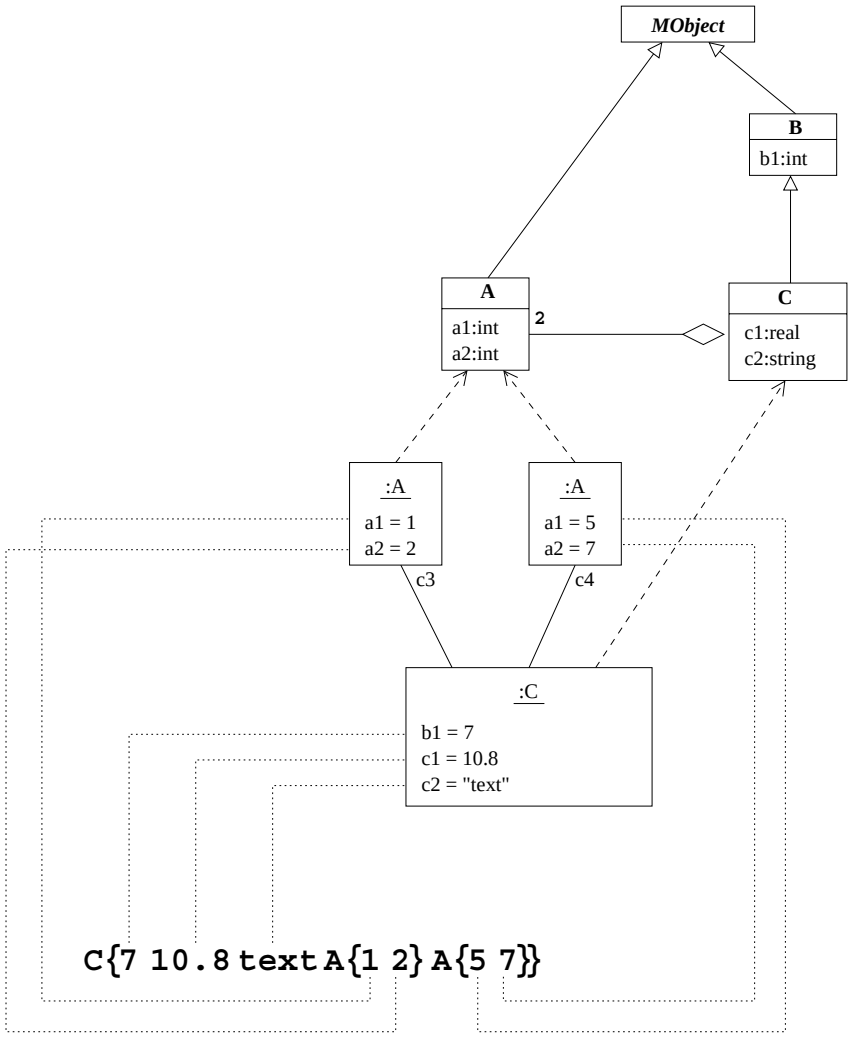

Figure 7.4: Object relations for *MCL* example.

`AddClassName()`

is very important and must be included in every constructor to make it's class name available for the event system. The next line of source code

```
b1 = arg.P->ReadInt()
```

reads an integer value from the `Parser arg.P`. This results in `b1` having a value of `7`. Going back to the constructor of `C` now the following text is still unread.

```
10.8 text A{1 2} A{5 7}
```

The first three lines should be clear. They simply read in a `real` value and a `string`. After these lines `c1` has a value of `10.8` and `c2` contains `"text"`. The next line

```
c3 = BuildObject(c3, arg);
```

needs more explanation. `BuildObject` is a member of the class MObject. The `BuildObject` method will build an object of any class derived from `MObject` in the same manner the construction of `C` originally started. The `ConstructArg` passed to a constructor of an `MObject` will be created by `BuildObject`. It is is a template method of `MObject` and can be called from within any `MObject` derived constructor. The template argument is the class which shall be constructed and the other argument is the `ConstructArg` of the calling constructor. A look at the constructor of `A` (see listing 7.3, line 1) should illustrate this. Here `arg.P` contains the following text:

```
1 2
```

`A`'s constructor works in the same way as the other examples. First of all the class-name is registered and then the two member variables `a1` and `a2` are read from the `Parser`. Going back to `C::C(carg_t arg)` the member `c3` has been initialized, with `c3->a1` equal to `1` and `c3->a2` equal to `2`. The next line in `C`'s constructor will do exactly the same for the member `c4`. Now the object has been completely constructed. The dotted lines in figure 7.4 show how the items of the *MCL* command-line map to the different members of the resulting objects.

7.2 Implementation of New Control-Commands

To illustrate how the existing control language can be extended, a few simple example commands shall be exemplary implemented. The goal is to implement an interpreter to compute simple mathematical functions. The following criteria have to be fulfilled:

- Functions can be used as input to other functions, in order to create nested expressions.
- Results need to be printed on the screen.
- An interface to the existing XY-plotting mechanism has to be provided.

The following simple functions serve as examples here:

- **Sum** Summation of two other functions.
- **Product** Multiplication of two functions.
- **Power** One function to the power of another one.
- **Sinus** The sinus of a function.

As base for these functions serves a class named `IntegralScalarValue`. Normally it is used to keep track of integral values (e.g. lift and drag of an airfoil). Indirectly it is a child class of `Action` and hence represents an executable command (see figure 7.5). The method **virtual void** `Action::Act()` is the interface for executable *MCL* commands. This method will be called, whenever an *MCL* command is invoked, thus it is the method which has to be overwritten to define the functionality of a new command. The idea is,

```
template <int N>
class Function : public IntegralScalarValue
{
protected:
  vector<IntegralScalarValue*> argument;
public:
  Function(carg_t arg);
  virtual void Act();
  virtual real Compute() = 0;
};

template <int N>
Function<N>::Function(carg_t arg) :
  IntegralScalarValue(arg),
  MObject(arg)
{
  AddClassName();
  argument.resize(N);
  for (int i = 0; i < N; ++i) BuildObject(argument[i],arg);
};

template <int N>
void Function<N>::Act()
{
  for (int i = 0; i < N; ++i) argument[i]->Act();
  value = Compute();
};
```

Listing 7.4: Base class for function example.

that when executed the integral value will be computed. Later on other objects (e.g. XY-plotters) can make use of the computed value. This is already the required interface to the visualization tools. It is fairly simple, `IntegralScalarValue` has a protected field named `value` which is supposed to hold the computed value. Derived classes have to set this value in there `Act` method. Thus after command has been invoked the numerical value can be retrieved by other objects using the method `realIntegralScalarValue::Value()`. In the present example it will be the result of a mathematical function. The functions mentioned above can take different numbers of arguments. Therefore a parameterized base-class `Function` is created with the template-parameter being the number of arguments the function can take. In this example the method `Act` will cause all `N` argument objects to execute. After this has been done the numerical values for the arguments are accessible and the abstract method `real Function<N>::Compute()` will be called. It has to be overwritten for concrete implementations of mathematical operators. The only thing to be done to implement a new function is to overwrite this method (see listing 7.4 and figure 7.5). The example implementation of a sum can be used to clarify this (see listing 7.5). The main requirement for an argument to a `Function` is that it is an object of the class `IntegralIsoValue`. Since `Function` is derived from that class, objects of the class `Function` can be used as arguments themselves. The main task of the constructor

```
struct Sum : public Function<2>
{
  Sum(carg_t arg);
  virtual real Compute();
};

Sum::Sum(carg_t arg) :
  Function<2>(arg),
  MObject(arg)
{
  AddClassName();
};

real Sum::Compute()
{
  return ( (argument[0]->Value()) + (argument[1]->Value()) );
};
```

Listing 7.5: Example function taking two arguments.

is to create the argument objects, which is done by calling **void** `BuildObject` (see listing 7.4, line 19) `N` times, where `N` is the template argument of the class `Function`. Pointers to the created objects will be kept in `vector<IntegralScalarValue*> argument`. The following mathematical expression shall be analyzed now:

$$y = \sin(10\,x) \cdot \frac{1}{10}\,x^2 \quad , \quad x \in \left\{-10, -9\frac{99}{100}, -9\frac{98}{100}, -9\frac{98}{100}, \ldots, 9\frac{99}{100}, 2\right\} \tag{7.1}$$

One thing, which is missing yet, is a way to specify an input. Till now, functions can accept other objects of the class `IntegralScalarValue` as input. They will not accept a real number as argument directly. To solve this problem a very simple constant-value class (see listing 7.6) will be used. Objects of this class will read in a real-value from the control-file and use this as output. The output will never change, no matter how often it will be called. Furthermore the variable x has a special role in this case. To create an XY-plot of a given function, the expression has to be evaluated for a set of discrete points. In order to do that a special function `RunningValue` will be implemented. This function takes two arguments:

1. a starting value
2. a discrete increment

A `RunningValue` will store its output-value and will increment it after every invocation (see figure 7.5). The example expression has two x'es. Of course it would be possible to use two `RunningValues` to express this. This is, however, error-prone, because it would be possible to provide different starting values or increments. Mistakes of this type would not be detected by the system. Still, they would lead to wrong results. To overcome this problem a simple class can be created, that copies the result from a function or a

```
class Constant : public IntegralScalarValue
{
private:

  real constant_value;

public:

  Constant(carg_t arg);
  virtual void Act() { value = constant_value; };
};

Constant::Constant(carg_t arg) :
  IntegralScalarValue(arg),
  MObject(arg)
{
  AddClassName();
  constant_value = arg.P->ReadReal();
};
```

Listing 7.6: A constant input-value.

scalar-value and uses this as output-value itself. `TakeOtherValue` is an example of such a class (see listing 7.7). Looking at the implementation (see listing 7.7) it can be seen how references to other control-language objects can be created (16). Listing 7.9 shows how the new commands can be put together in order to get an XY-plot of the above function. `$PLOT` is a control-file variable pointing to an XY-plot window. This would have to be created somewhere before. Finally figure 7.6 shows the object-tree which would be created by this example. In figure 7.7 the same object-tree can be found, but with the `MObject` related associations shown. Every control file that is interpreted will create an object-tree of this type and it will be very easy to find other objects of certain classes in this tree.

```
class TakeOtherValue : public IntegralScalarValue
{
  IntegralScalarValue *other_value;

public:

  TakeOtherValue(carg_t arg);
  virtual void Act() { value = other_value->Value(); };
};

TakeOtherValue::TakeOtherValue(carg_t arg) :
  IntegralScalarValue(arg),
  MObject(arg)
{
  AddClassName();
  ResolveVar(other_value,arg);
};
```

Listing 7.7: A reference to another value as input.

```
int main(int argc, char **argv)
{
  MouseMain mouse;
  MObject *top = MouseMain::Top();
  QtExtensions::Register();
  new MObjectBuilder<Constant>   (top, "Constant");
  new MObjectBuilder<Sum>        (top, "Sum");
  new MObjectBuilder<Sinus>      (top, "Sinus");
  new MObjectBuilder<Power>      (top, "Power");
  new MObjectBuilder<Product>    (top, "Product");
  new MObjectBuilder<RunningValue> (top, "RunningValue");
  new MObjectBuilder<TakeOtherValue> (top, "TakeOtherValue");
  mouse.StartMouse(argc, argv);
};
```

Listing 7.8: Registering the new commands.

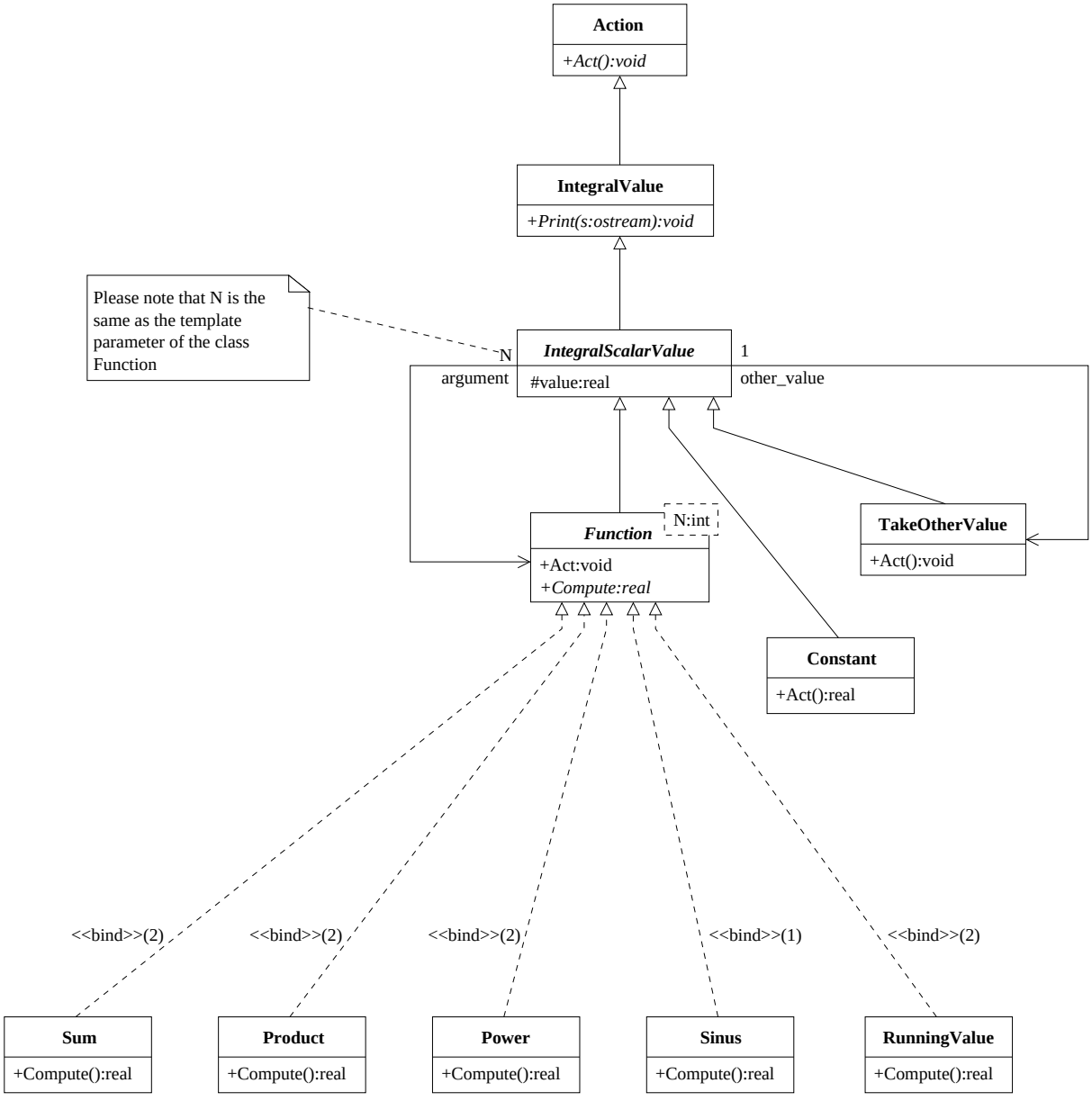

Figure 7.5: Class diagram for example *MCL* commands.

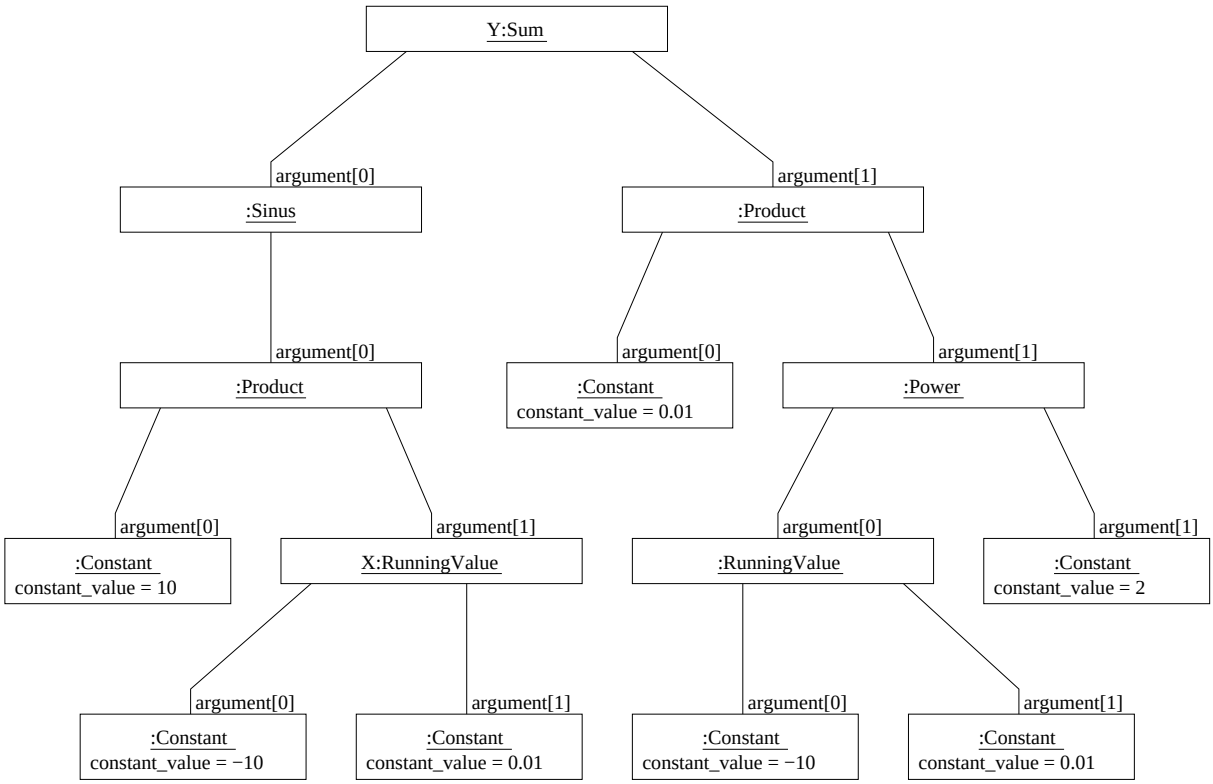

Figure 7.6: Instances for function-plotting example.

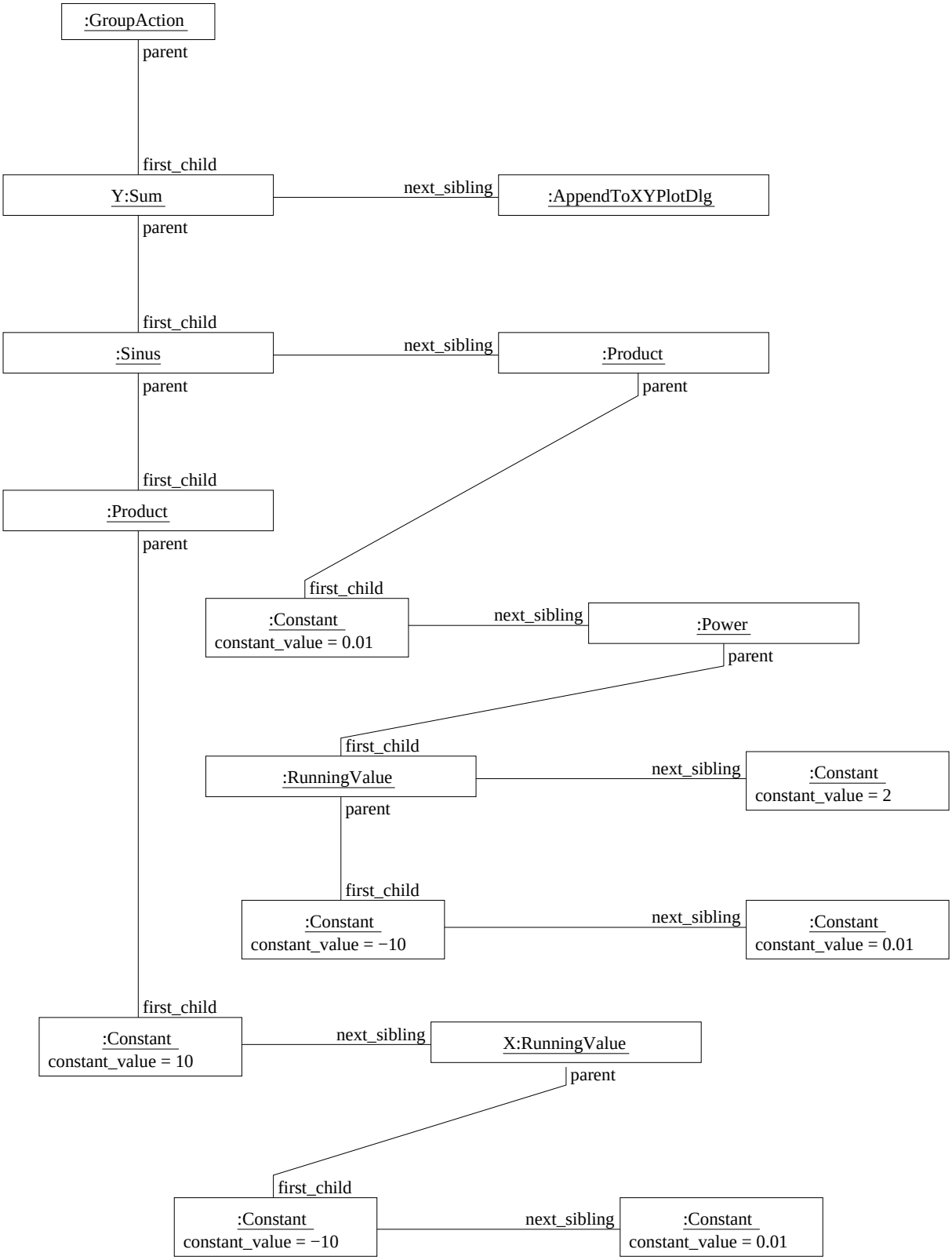

Figure 7.7: Relations of function-plotting instances as *MOUSE* classes.

```
StepAction{
  GroupAction {
    %Y Sum{
      Sinus{
        Product{
          Constant{10}
          %X RunningValue{
            Constant{-10}
            Constant{0.01}
          }
        }
      }
      Product{
        Constant{0.1}
        Power{
          TakeOtherValue{$X}
          Constant{2}
        }
      }
    }
    AppendToXYPlotDlg{$PLOT $X $Y}
  }
  2000
}
```

Listing 7.9: Control-file for function plotting.

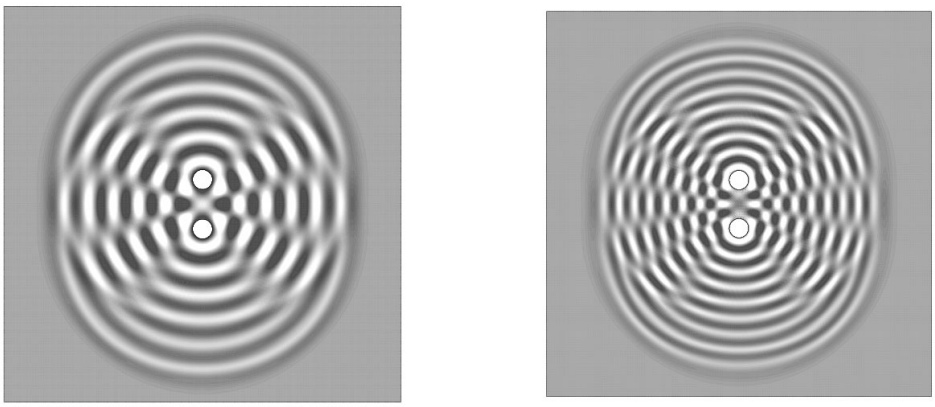

Figure 7.8: Wave-patterns for different frequencies.

7.3 Implementation of New Numerical Operators

7.3.1 Solving the Linear Wave Equation in 2D

As an example for the integration of a new equation into the library, the scalar wave equation

$$\frac{\partial^2 \phi}{\partial t^2} = c^2 \nabla^2 \phi \tag{7.2}$$

shall be realized. c is the propagation speed. The equation shall only be solved for two-dimensional problems in this simple example. For a finite-volume discretization the

```
inline void AdvanceWaveEquation::operate(size_t i_tmp, size_t i)
{
  real delta_t = cfl->CFL()*cfl->TimeStep(i);
  real u = delta_t*delta_t*PL->IVol(i)*res_u[i] + 2*u_new[i] - u_old[i];
  u_old[i] = u_new[i];
  u_new[i] = u;
};
```

Listing 7.10: Time-step routine for linear wave propagation.

```
WaveOperator2D::WaveOperator2D(carg_t arg) : Operator(arg)
{
  AddClassName(); ResolveVar(PARA,arg);
  a = PARA->RealParam("wave-speed");
  u = VL->Get1DVar("new", "u");
  du_dx = VL->Get1DVar("gradx", "u");
  du_dy = VL->Get1DVar("grady", "u");
  res_u = VL->Get1DVar("res", "u");
};
```

Listing 7.11: Constructor of base-class for wave propagation operators.

integral form of that equation is more appropriate:

$$\int_V \frac{\partial^2 \phi}{\partial t^2} \mathrm{d}V = c^2 \oint_{\partial V} \nabla \phi \, \mathbf{n} \, \mathrm{d}A \,. \tag{7.3}$$

A fairly simple discretization is to be used for this example on the implementation of new numerical operators. The right hand side of the above equation for a discrete node i shall be discretized in the following form:

$$\left[c^2 \oint_{\partial V} \nabla \phi \, \mathbf{n} \, \mathrm{d}A \right]_i = \mathrm{Res}(\boldsymbol{\phi})_i \quad , \quad \mathrm{Res}(\boldsymbol{\phi})_i = c^2 \sum_{j=1}^{N_{\mathrm{ngh},i}} \left((\boldsymbol{\nabla \phi})^{\leftrightarrow}_{(i,j)} \cdot (\mathbf{n} \Delta A)_{(i,j)} \right) . \tag{7.4}$$

$(\mathbf{n}\Delta A)_{i,j}$ is directly available from the `PopList2D`. The value $(\boldsymbol{\nabla \phi_{i,j}})^{\leftrightarrow}_{i,j}$ is a centrally reconstructed gradient on the edge (i,j). Practically this can be done by simply averaging the gradients on the nodes i and $\{i,j\}$. The gradient $(\nabla\phi)_i$ is available on every node of the mesh. It can be computed beforehand, using `ComputeFLSQGrad2D` for example. This would be a least-squares approach and it shall be used in this case. To get the gradient on an edge a simple arithmetic averaging shall be performed:

$$\mathrm{Res}(\boldsymbol{\phi})_i \; = \; c^2 \sum_{j=1}^{N_{\mathrm{ngh},i}} \left(\frac{1}{2} (\boldsymbol{\nabla \phi}_i + \boldsymbol{\nabla \phi}_{\{i,j\}}) (\mathbf{n} \Delta A)_{(i,j)} \right) . \tag{7.5}$$

```
inline void WaveFlux2D::operate(size_t ie, size_t ip, size_t from, size_t to,
                                real nx, real ny, bool mod_from, bool mod_to)
{
  real du_dx_ave = 0.5*(du_dx[from]+du_dx[to]);
  real du_dy_ave = 0.5*(du_dy[from]+du_dy[to]);
  real flux = -a*a*(du_dx_ave*nx + du_dy_ave*ny);
  if (mod_from) res_u[from] -= flux;
  if (mod_to) res_u[to] += flux;
};
```

Listing 7.12: Spatial operator for wave propagation.

For left side of the equation the following three-level scheme will be used:

$$\left[\int_V \frac{\partial^2 \phi}{\partial t^2} \mathrm{d}V\right]_i = \frac{(\Delta V)_i}{\Delta t^2}(\phi_i^{n-1} - 2\phi_i^n + \phi_i^{n+1})\,. \tag{7.6}$$

V_i is the size of the discrete control-volume. With (7.4), (7.5) and (7.6) the complete discretized equation for a node i can be written as:

$$\frac{(\Delta V)_i}{\Delta t^2}(\phi_i^{n-1} - 2\phi_i^n + \phi_i^{n+1}) = c^2 \sum_{j=1}^{N_{\mathrm{ngh},i}} \left(\frac{1}{2}(\boldsymbol{\nabla}\boldsymbol{\phi}_i + \boldsymbol{\nabla}\boldsymbol{\phi}_{\{i,j\}})(\mathbf{n}\Delta A)_{(i,j)}\right). \tag{7.7}$$

If the above equation is solved for ϕ_i^{n+1} a time-advancing scheme can be formulated:

$$\phi_i^{n+1} = \frac{\Delta t^2}{V_i}\mathrm{Res}(\phi^n)_i + 2\phi_i^n - \phi_i^{n-1}. \tag{7.8}$$

Please note, that only two time-levels need to be stored. Listing 7.10 shows the implementation of the time advancing scheme. `cfl` is a pointer to a `CflManager`, a class which is normally used to handle time-step restrictions due to numerical stability criteria. In this particular case the time-step will simply be set to a constant value. `CflManager`, however, still provides a useful interface to other objects. `PL` is a pointer to the `PointList` and it is used to access the inverse control volume of the node `i`, which is a parameter for the `operate` method. The variables `u_new`, `u_old` and `res_u` are automatic pointers. The field named "res" is the one which is used to cumulate the right hand side of the equation. The discretization for Res is very simple. Since the gradients are available on every node, only a simple operator has to be created that computes the mean value and the scalar product on an edge. Listing 7.12 shows how it can be implemented. The constructor of the common base-class can be found in listing 7.11. Figure 7.8 shows two simulations of two interacting exciters. These simulations have been made, using the code introduced in this section.

7.3.2 Simulating a Non-Linear Convection Problem in 3D

The following equation in integral form

$$\int_V \frac{\partial \mathbf{q}}{\partial t}\mathrm{d}V + \oint_{\partial V} \mathbf{H}\,\mathbf{n}\,\mathrm{d}A = 0 \quad , \quad \mathbf{q} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} \quad , \quad \mathbf{H} = \mathbf{q} \otimes \mathbf{q} \tag{7.9}$$

shall be solved, using an upwind scheme plus Runge Kutta time-stepping. $(\mathbf{v} \otimes \mathbf{v})$ is a dyadic product and it evaluates to:

$$(\mathbf{v} \otimes \mathbf{v}) = \begin{pmatrix} u^2 & uv & uw \\ uv & v^2 & vw \\ uw & vw & w^2 \end{pmatrix}. \tag{7.10}$$

The divergence form of the integral equation (7.9) can be written as follows:

$$\begin{aligned} & \frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(u^2) + \frac{\partial}{\partial y}(uv) + \frac{\partial}{\partial z}(uw) = 0 \\ \wedge\ & \frac{\partial v}{\partial t} + \frac{\partial}{\partial x}(uv) + \frac{\partial}{\partial y}(v^2) + \frac{\partial}{\partial z}(vw) = 0 \\ \wedge\ & \frac{\partial w}{\partial t} + \frac{\partial}{\partial x}(uw) + \frac{\partial}{\partial y}(vw) + \frac{\partial}{\partial z}(w^2) = 0\,. \end{aligned} \tag{7.11}$$

To solve (7.9) numerically, using the available finite-volume library, a discrete approximation for $\mathbf{H}$ has to be found. If formulated across a discrete interface, the flux vector on an interface i, j becomes:

$$\begin{aligned} \left[\mathbf{H}(\mathbf{n}\Delta A)\right]_{(i,j)} &= \left(\tilde{\mathbf{v}}_{(i,j)} \otimes \tilde{\mathbf{v}}_{(i,j)}\right)(\mathbf{n}\Delta A)_{(i,j)} \\ &= \left((\mathbf{n}\Delta A)_{(i,j)} \cdot \tilde{\mathbf{v}}_{(i,j)}\right) \cdot \tilde{\mathbf{v}}_{(i,j)} \end{aligned} \tag{7.12}$$

To construct an upwind discretization, the flux can be split into a scalar propagation speed and a vector to be propagated. The term $(\mathbf{n}\Delta A)_{(i,j)} \cdot \tilde{\mathbf{v}}_{(i,j)}$ from (7.12) can be interpreted as scalar propagation speed across the interface (i, j). This term can be discretized centrally or by using one-sided values. The following discretization uses a weighting factor to switch between a central or an upwind propagation speed.

$$\begin{aligned} \left[\mathbf{h}(\mathbf{n}\Delta A)\right]_{(i,j)} = \Delta A \left(\omega\, \tilde{c}^{\leftrightarrows}_{(i,j)} + (1-\omega)\tilde{c}^{\leftrightarrow}_{(i,j)}\right) \tilde{\mathbf{v}}^{\leftrightarrows}_{(i,j)} \quad , \quad \tilde{c}^{\leftrightarrow}_{(i,j)} &= \mathbf{n}_{(i,j)} \cdot \tilde{\mathbf{v}}^{\leftrightarrow} \\ \tilde{c}^{\leftrightarrows}_{(i,j)} &= \mathbf{n}_{(i,j)} \cdot \tilde{\mathbf{v}}^{\leftrightarrows} \end{aligned} \tag{7.13}$$

The superscript $^{\leftrightarrow}$ denotes a central value, whereas $^{\leftrightarrows}$ denotes a one-sided value, depending on the direction of $\mathbf{v}^{\leftrightarrow}$. Please note that the variables, which will be propagated, are upwinded, no matter which value the weighting factor ω will be set to. Thus this discretization is an upwind one and it will be conditionally stable, depending on the *CFL* number and the Runge-Kutta coefficients used.
Figure 7.9 shows a few intermediate states of such a non-linear convection simulation. The computational domain is a cube, which has been meshed, using a simple Cartesian grid (70,602 elements). As initial condition $\mathbf{v} = \mathbf{0}$ has been used for the whole domain, except for the boundaries. On the boundaries the convection velocity has been set to $\mathbf{v} = (1/|\mathbf{n}|)\mathbf{n}$, with $\mathbf{n}$ as the local normal vector of the boundary. In figure 7.9 stream lines can be seen, as well as the iso-surface $|\mathbf{v}| = 10^{-1}$.

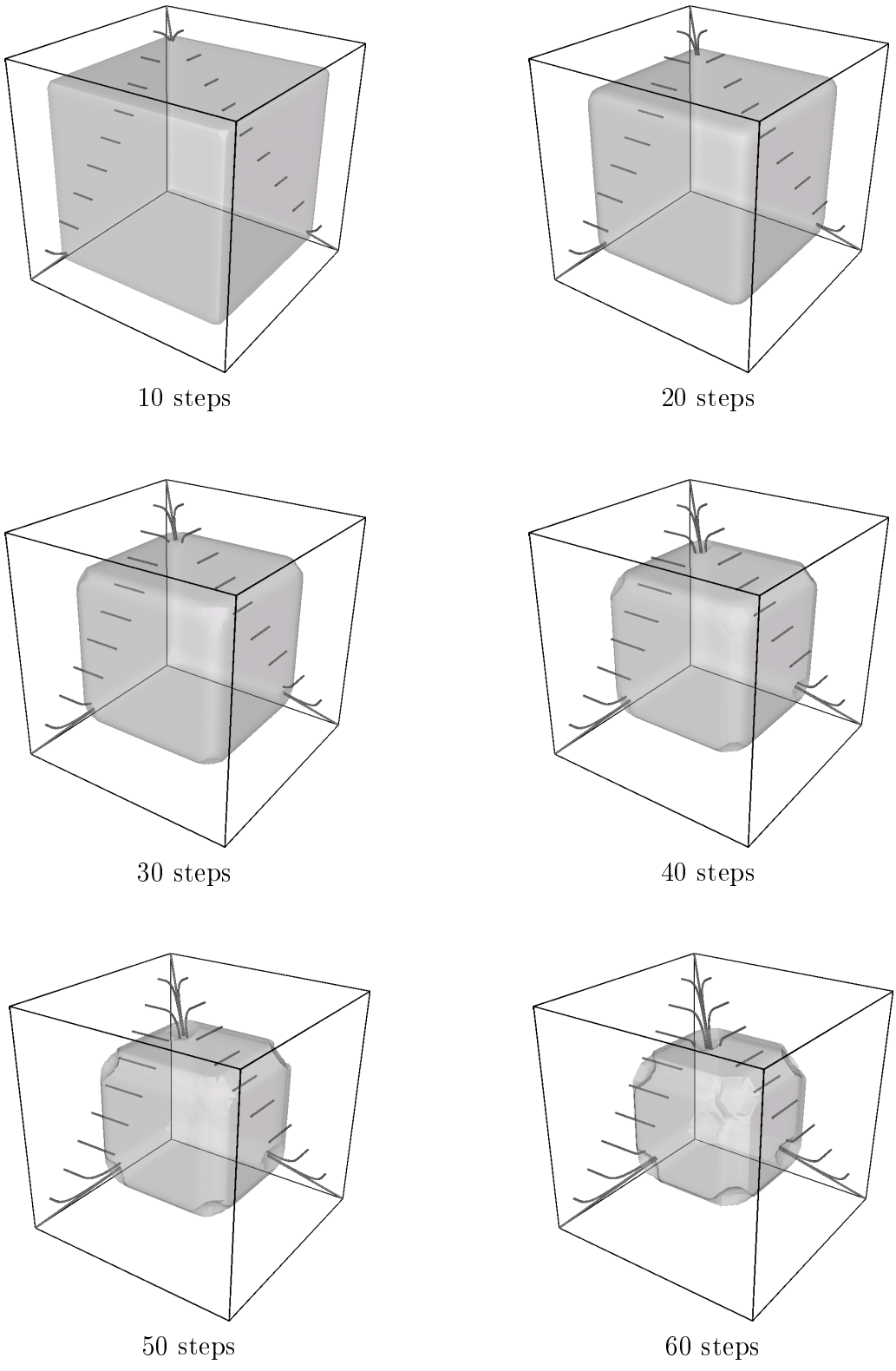

Figure 7.9: Numerical simulation of a non-linear convection problem.

```
inline void BurgersFlux3D::operate(size_t ie, size_t ip, size_t from, size_t to,
                                   real nx, real ny, real nz, bool mod_from, bool mod_to)
{
  real vn = u_ave[ie]*nx + v_ave[ie]*ny + w_ave[ie]*nz;
  real flux_u, flux_v, flux_w;
  real U, V, W;
  if (vn > 0) {
    U = u_from[ie]; V = v_from[ie]; W = w_from[ie];
  } else {
    U = u_to[ie]; V = v_to[ie]; W = w_to[ie];
  };
  flux_u = omega*(U*U*nx + U*V*ny + U*W*nz) + (1-omega)*(vn*U);
  flux_v = omega*(U*V*nx + V*V*ny + V*W*nz) + (1-omega)*(vn*V);
  flux_w = omega*(U*W*nx + V*W*ny + W*W*nz) + (1-omega)*(vn*W);
  if (mod_from) {
    res_u[from] -= flux_u;
    res_v[from] -= flux_v;
    res_w[from] -= flux_w;
  };
  if (mod_to) {
    res_u[to] += flux_u;
    res_v[to] += flux_v;
    res_w[to] += flux_w;
  };
};
```

Listing 7.13: Flux operator for non-linear convection.